Java 弱引用

https://droidyue.com/blog/2014/10/12/understanding-weakreference-in-java/

- weakWidget.get() 就可以获取到真实对象, 当没有任何强引用到 widget 对象时，get会返回 null
- WeakHashMap 的 Key 是弱引用，被回收时整条记录会被自动移除, 可以转换为 HashMap
- 在构造 WeakReference 时可以传入 ReferenceQueue 对象，当该引用指向的对象被标记为垃圾时，弱引用会自动加入队列，我们可以定期清理队列
- 弱引用可达会被回收，软引用可达会在内存不足时回收，适合做缓存
- 弱引用指向对象被标记为垃圾后，弱引用就加入了引用队列，get 就会返回 null，但垃圾对象可以复活，如用finalize方法创建强引用指向它，但虚引用只有在对象被回收之后才会加入引用队列，get 才会返回 null，它允许你知道具体何时对象从内存中移除

启动模式

https://juejin.im/post/59b0f25551882538cb1ecae1

https://blog.csdn.net/zhangjg_blog/article/details/10923643

- singleTop:栈顶的走一遍 onNewIntent
- singleInstance: 单独一个栈，再次启动走一遍 onNewIntent（**全局唯一性**）
- singleTask: Activity 只存在 taskAffinit 任务栈中，同一时刻只有一个实例，再次唤醒清除栈顶，回调 onNewIntent（**任务栈内唯一性**）
- 存放singleInstance Activity实例的Task只能存放一个该模式的Activity实例。如果从singleInstance Activity实例启动另一个Activity，那么这个Activity实例会放入其他的Task中。同理，如果singleInstance Activity被别的Activity启动，它也会放入不同于调用者的Task中。
- 上面场景只适用于 Activity 启动 Activity, 并且默认都是 Intent, 没有额外添加任何 Flag 的情况
- taskAffinity 可以跨 app 使用，让不同 app 的 singleTask 模式的在一个任务栈中
- 在启动一个 singleTask 的 Activity 实例时，如果系统中已经存在这样一个实例，就会将这个实例调度到任务栈的栈顶，并清除它当前所在任务中位于它上面的所有的 activity。
- NEW_TASK 先检测前台 Task 是否与 taskAffinity 对应，不对应则移到前台或者创建，然后判断是否为 root Activity（栈底），不是则新建 ac。
- 非 Activity 启动 Activity 要强制规定使用 NEW_TASK, 源码中 ContextImpl 做了检查，没有该 Flag 会抛出异常，其实新 Task 是为了保证 APP 的独立性，将不同业务的 Activity 限制在自己的任务栈内，back 时符合逻辑
- action 字段作用是匹配所有可能的，只要 filter 中包含该字段就可被启动，category 字段作用是加以限定只有 filter 中；如果 Intent 中包含 c 字段，那么这写 c 字段要包含于 filter 中的 c 字段才行；data 由 mimeType 和 URI 组成.

Java 入门细节

https://www.jianshu.com/p/fabad9250b1b

- JRE：所有程序都要在 JRE 下运行。JRE 包括 JVM。
- JVM：是 JRE 的一部分，解释自己的指令集（字节码文件）映射到 CPU 指令集。
- JDK：物理存在是 JRE，JVM，programming tools 的集合

- 启动一个 Java 程序，一个 JVM 实例就产生，main() 函数作为程序起点，是非守护线程；由守护线程启动的都是守护线程，当不存在非守护线程时，守护线程全部结束；**最常见的守护线程是 GC**
- 当所有非守护线程都终止时 JVM 才退出
- 类加载：loading（装载）将字节码载入 JVM，标识符为：类名+包名+ClassLoader实例ID; linking(链接)初始化类中的静态变量解析接口调用；initializing（初始化）执行类静态初始化代码
- PC，JVM 栈（存放局部变量），Heap（分为新生代和旧生代，线程共享），方法区（方法及static，final），常量池（空间从方法区分配），本地方法栈（支持native方法执行）

List

https://blog.csdn.net/myf0908/article/details/80608786

- ArrayList 超出容量就会扩容 50%，调用 System.copy(), 包括 add（i，e），remove（i），性能较差
- LinkedList 是双向链表, 因为 Node 本身使用了更多的空间，get，set 时 i 大于 index/2 会从尾部开始遍历
- Vector 所有的方法都是同步的，是线程安全的，ArrayList 是线程不安全的

App 启动

https://www.jianshu.com/p/e69d22ec0582

https://www.jianshu.com/p/12de32b31836

- 冷启动是指后台没有对应 App 的进程，需要创建进程，初始化 Application，创建和初始化 Launch Activity；热启动直接初始化 Launch Activity

> 一. Step1 - Step 11： Launcher通过Binder进程间通信机制通知 AMS， 它要启动一个Activity；
>
> 二. Step 12 - Step 16： AMS 通过 Binder 通知 Launcher 进入 Paused 状态；
>
> 三. Step 17 - Step 24： Launcher 通过 Binder 通知 AMS，它已经准备就绪进入Paused状态， 于是 AMS 就创建一个新的进程，用来启动一个 ActivityThread 实例， 即将要启动的 Activity 就是在这个 ActivityThread 实例中运行；
>
> 四. Step 25 - Step 27： ActivityThread 通过 Binder 进程间通信机制将一个 ApplicationThread 类型的 Binde r对象传递给 AMS， 以便以后 AMS 能够通过这个 Binder 对象和它进行通信；
>
> 五. Step 28 - Step 35：AMS 通过 Binder 进程间通信机制通知 ActivityThread， 现在一切准备就绪，它可以真正执行Activity的启动操作了。

- AMS 是从 Zygote 进程中 fork 一个新的进程给应用的 `Zygote`：后面再说吧
- 点击 icon，Launcher 会调用 context.startActivity(intent)
- AMS 是一个守护进程，运行在 Android Framework 中，为了减少用户处理 IPC 问题，就有了 ActivityManager 作为中介
- AMS 收到 Launcher 通知后，他会先收集 intent，然后根据 uri 检查是否有权限启动指定 Activity，如果有 检查 ProcessRecord 如果不为 null，就是热启动，否则创建新进程
- `adb shell am start -W [packageName]/[packageName.launchActivity]` 计算 app 启动时间，或者通过打桩
- 避免 Application 中的耗时操作，减少 Launch Activity 的 View 层级，白屏是因为 Activity 已进入，但未加载到布局文件，就显示了 window 窗口的背景，可以修改主题中 window 窗口的背景图

Launcher

https://www.cnblogs.com/tiantianbyconan/p/5017056.html

- Launcher 也是一个 Activity，在 onResume（）mDesktopLoaderThread 中加载 app 信息
- icon 绑定时，APPLICATION 和 SHORTCUT 都进入到 shortcut case，调用 createShortcut()
- createShort() 中会 inflate；注册 Click，通过 ApplicationInfo 获取 Intent, 调用 startActivitySafely 其中添加了 NEW_TASK FLAG
- `public ActivityResult execStartActivity ()` 中 `ActivityManagerNative.getDefault()` 获得了一个 IActivityManager 的实现类，通过 `ActivityManagernative` 封装为 `ActivityManagerProxy` 而后调用 AMS 的 startActivity
- `startSpecificActivityLocked` 中会检查 ProcessRecord，如果为 null 则 `mService.startProcessLocked` 否则 `realStartActivityLocked`
- 在 `startProcessLocked` 方法中会根据 ApplicationInfo，uid 创建新的 ProcessRecord，该函数有一个参数 *entryPoint* 默认为 “android.app.ActivityThread” 作为创建的进程的主入口，会调用 main 方法
- 接下来就是通过 Zygote fork 一个新的 app 进程，然后就进入了 main 方法
- 在 `main` 方法中创建了 ActivityThread，调用了 `attach` 方法传入 ApplicationThread 作为 binder, 而后启动了 Looper
- AMS 收到 ApplicationThread 后经过一系列回调会执行 ActivityThread 的 `handleBindApplication` ，其中先创建了当前 app 的 Context, 而后初始化 Instrumentation 和 Instrumentation 的 Context， 然后创建 Application 实例，调用 onCreate
- 而后 AMS 就会从栈顶获取要启动的 Activity，通过 binder 然后进入 AtivityThread 的 `scheduleLaunchActivity` 方法中，然后通过 `Handler` 进入 `handleLaunchActivity` 方法
- 先调用 `performLaunchActivity` 方法返回一个 `Activity` ，然后调用 `handleResumeActivity` 方法让该 `Activity` 进入 `onResume` 状态。所以很显然在 `performLaunchActivity` 中肯定是生成了 `Activity` 实例，并调用了 `onCreate` 方法了（详细的看 performLaunchActivity，可以发现 System/App/Activity Context 都是通过 ContextImpl 生成的）
- Activity 的 attach 方法与 ACThread Instrumentation 进行了绑定，所以每个 Activity 都含有一个 ACThread 和 Ins 实例

Set

- HashSet 的内部就是 HashMap，因为 Map 里的 KeySet 就是一个 Set，而 value 是假值，全部使用一个 Object 即可
- 根据 hascode 计算位置， 位置相同的根据 equals 判断是否相同，相同则重复否则都存在同一个桶区内
- LinkedHashSet：内部是LinkedHashMap，每个 Entry 作为节点组成双向链表
- TreeSet：内部就是 TreeMap 的 SortedSet，不是根据 equals 而时根据 Comparable 或者是传入的 Comparator 来确定，当 compareTo 返回 0 是则说明重复
- 上面所说的均为线程不安全的

---

volatile & final & transient

- volatile 首先本身包含进制指令重排序的语义，不会将该变量上的操作与其它内存操作一起重排序，通过内存屏障实现（后面的代码不会排序到屏障之前）
- volatile 保证了改变量对所有线程的可见性，任何修改其它线程均可见，会使其它线程中改变量从内存中拷贝的值无效，需要从新读取

- final 可以修饰变量，形参，表示变量不可修改，final 修饰方法表示不可被重写，修饰类表示不可被继承
- transient 修饰变量，表示该类在实现序列化时，改变量不需要保存在磁盘中

Parcelable & Serializable

https://blog.csdn.net/xia_leon/article/details/83278452

- Serializable 是一个 mark Interface，无需实现方法，Java 便会对这个对象进行高效的序列化操作，但是使用了反射，效率较低
- Parcelable 需要重写 writeToParcel 方法，而后创建 static Parcelable.Creator 实现反序列化，write 和 read 时要保证顺序相同
- 可序列化对象的成员变量也要是可序列化的

Lock

https://www.cnblogs.com/handsomeye/p/5999362.html

- 当读读不冲突而读写冲突时 synchronized 关键字就会导致效率低下
- Lock 需要手动释放锁，且发生异常时也不会释放，需要在 catch finally 中手动释放
- tryLock 会立即返回获取锁的结果，可以传如一定的等待时间；lockInterruptibly 方法获取锁时如果处于等待状态，则可以被 interrupt 中断，并抛出异常。而 synchronized 处于等待状态时是无法被中断的
- 注意，如果需要正确中断等待锁的线程，必须将获取锁放在 try 外面，然后将InterruptedException 抛出
- ReadWriteLock 的实现类 ReentrantReadWriteLock 可以分为读锁和写锁，读读之间不冲突，但读写，写写之间会等待

Lock 锁

https://www.cnblogs.com/handsomeye/p/5999362.html

- 重进入锁，也就是 Java 的重进入机制，锁的分配基于每线程而不是每次，每个锁具有一个计数器，当计数器为 0 或者申请线程为持有该锁的线程时，申请成功计数器加一，不会造成同锁方法互调死锁
- 可中断锁，synchronized 是不可中断，lock 是可中断锁
- 公平锁，尽量按请求顺序来获取锁，ReentrantLock 和 ReentrantReadWriteLock 默认非公平，但是可以设置为公平锁
- 读写锁 - ReadWriteLock （Interface）

---

synchronized

https://www.cnblogs.com/jiangds/p/6476293.html

- 多个线程访问同一个监视器的时候，监视器会将这些请求存储在不同容器中
- >Contention List：竞争队列，所有请求锁的线程首先被放在这个竞争队列中
- Entry List：Contention List中那些有资格成为候选资源的线程被移动到Entry List中
- Wait Set：哪些调用wait方法被阻塞的线程被放置在这里
- OnDeck：任意时刻，最多只有一个线程正在竞争锁资源，该线程被成为 OnDeck
- Owner：当前已经获取到所资源的线程被称为Owner

- !Owner：当前释放锁的线程
- 锁消除，编译器在运行时，基于逃逸分析，检测到共不可能存在共享数据竞争的锁进行消除
- 自旋锁，本质是执行几个空方法，循环或者几行空汇编指令
- 轻量级锁，会在当前线程的栈中建立一个拷贝 Lock Record 并尝试指向它，对于绝大部分的锁，在整个同步周期内都是不存在竞争的，轻量锁使用 CAS 操作消除了同步使用的互斥量的开销
- 偏向锁，本质是一个变量，判断这个变量是否是当前线程，就是避免加锁解锁操作

HashMap

https://yikun.github.io/2015/04/01/Java-HashMap工作原理及实现/

https://www.jianshu.com/p/1e9cf0ac07f4

- 是基于Map接口的实现，存储键值对时，它可以接收null的键值，是非同步的，HashMap存储着Entry(hash, key, value, next)对象。
- 超过 load factor（0.75），capacity 扩容一倍
- 对 key 的 hashcode（）做 hash，再计算 index
- equals（）方法是为了在 putVal，如果节点已经存在，判断是否为碰撞的依据，equals相同就代表是同一个 key
- 链表长度超过 TREEIFY_THRESHOLD（8）会转换为红黑树
- get 时，不停的用 equals 方法判断，直到命中
- `return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16)` hash 函数，高16不变，低16与高16异或，保证 n 比较小的时候高位也能参与运算，且异或开销小
- get 可能形成死循环是因为向新表中拷贝时是头插法，如果多线程会导致出现死循环，将新表中的后一个元素头插回头部，因为另一条线程 e 和 next 没有及时更新的缘故

ConcurrentHashMap

https://www.cnblogs.com/ITtangtang/p/3948786.html

http://www.importnew.com/28263.html

- 内部是一个 Segment 数组，Segment 继承了 ReentrantLock 加锁
- concurrencyLevel 不可扩容默认为 16，可以理解为有 16 个 Segments，只要操作分布在不同的 Segment 上就可以最多同时支持 16 个线程并发写
- 表的长度会根据初始容量设置为最近的 2的幂次，原因： & 比 % 操作快，扩容方便
- hash 值的最高 4 位决定分配在哪个 Segment 中
- entry 中只有 value 字段是 volatile（可以保证在 get 时不需要加锁，同时还需要 segment 中 volatile count 保证），其它字段包括 next 均为 final，添加时只能在头部添加，删除时需要重新拷贝被删除节点前面的部分
- 统计 size 时，会先不加锁累加 count 求和两次，根据 modcount 是否改变来判断容器是否发生变化，如果发生变化则对所有的 put，remove 加锁后重新统计

https://blog.csdn.net/zguoshuaiiii/article/details/78495332

- Java8 中取消了 segments 字段，直接采用 Node<K,V>[] table，同时也取消使用了 ReentrantLock，而是直接使用 synchronized 对链表的头结点加锁，可能是考虑到 Lock 的数量，同时改为先添加结点，添加后考虑是否需要扩容
- Java8 中 hash 值为 MOVED 表示正在扩容，可能会要求其它线程一起帮助扩容；sizeCtl 是一个控制符，代表这不同状态

- Map 中存的实际上是 TreeBin 数组，TreeBin 中包含 `TreeNode<K,V> root` ,该类结点 hash 值为 TREEBIN 常量，作为标识
- ForwardingNode 包含 `final Node<K,V>[] nextTable` ,find 方法是从 nextTable 中查
- 随处可以看到U, 大量使用了U.compareAndSwapXXX的方法，这个方法是利用一个CAS算法实现无锁化的修改值的操作，他可以大大降低锁代理的性能消耗。这个算法的基本思想就是不断地去比较当前内存中的变量值与你指定的一个变量值是否相等，如果相等，则接受你指定的修改的值，否则拒绝你的操作。因为当前线程中的值已经不是最新的值，你的修改很可能会覆盖掉其他线程修改的结果。这一点与乐观锁，SVN的思想是比较类似的
- 不允许 key 或 value 为 null
- transfer方法中在空结点上插入forward节点，如果 put 时检测到需要插入的位置被forward节点占有，就帮助进行扩容

View.Post

https://www.cnblogs.com/dasusu/p/8047172.html

- 既然可以对 UI 进行操作，那么本质上就是使用了 Handler 来实现
- 当 attachInfo 不为 null 时调用 attachInfo.mHandler.post(action), attachInfo 在 `dispatchAttachedToWindow` 和 `dispatchDetachedFromWindow` 中初始化和销毁
- 在 onCreate 中 attachInfo 为 null 所以会调用 `getRunQueue().post()` ,内部是 `HandlerActionQueue.post` ，是将我们的 runnable 包装为一个 Handler Action 放到数组里暂存起来，而回调方法 `executeActions(Hadnler handler)` 会将 action 解构后调用传入的 `handler.postDelay` ,继续跳转可以发现这个 Handler 是在 `dispatchAttachedToWindow` 中传过来的
- 同时我们发现在 ViewGroup 的 addViewInner 中 如果 attachInfo 非空会调用 `child.dispatchAttachedToWindow` ,并将 attachInfo 传入，所以为同一个 attachInfo，而我们发现 ViewGroup 的 dispatch 方法中会先调用 super.dispatch 即先作为一个 View 而后在遍历调用 child.dispatch, 而 child 的 attachInfo 又来自于 parent，那么要看一下最顶级 view 的 attachInfo 来自哪里
- 最终回到 `ViewRootImpl.performTraversals` 其中会调用 Ac 的 DecorView （作为View）的 dispatch 方法传入 attachInfo（从此就有了 attachInfo，后面就可以先从 ViewGroup 的开始遍历了），而 attachInfo 是在 ViewRootImpl 构造函数中创建的，创建时传入了一个 mHandler，每个 Handler 在创建时都会绑定一个 Looper, 该对象创建时调用无参构造函数，所以绑定的是当前 Looper，又 new 是在主线程所以绑定的是主线程 Looper
- 在调用 DecorView 的 dispatch 后（各 Runnable 已经加入到 Msg 队列）开始遍历 View 树，进行绘制，测量，完成后代表当前 Msg 完成，可以从 Looper 中获取下一个 Msg，这就可以保证 post 在绘测完成后执行
- mHandler 持有外部类 ViewRootImpl 引用，使用不当如任务延迟，可能导致外部类无法回收造成内存泄漏

泛型

https://blog.csdn.net/s10461/article/details/53941091

https://www.cnblogs.com/scutwang/articles/3735219.html

泛型方法可以类比 findViewById<>()

- 泛型只在编译期有效，在编译过程中正确检查泛型结果后会将泛型的相关信息擦出，并且在对象进入和离开的边界处添加类型检查和类型转换的方法，泛型类型实际上都是相同的基本类型
- 定义的泛型类不一定要传入泛型类的实参，如果传入会做相应限制，可以不传入，则对应的泛型方法，变量可以为任意类型

- 泛型类型不能是基本类型；不能对具体泛型类使用 instanceof `num instanceof Generic<Number>`
- 为了可以统一 Coll 和 Coll 可以写为 Coll <?> 此处 ？ 是类型**实参**，不是形参，表示任意类型。使用具体功能时，如果只使用 Object 中的功能可以使用 ？ 通配符。
- 只有在方法前面声明了 的才是泛型方法，在方法里使用了泛型类声明的泛型不算是泛型方法
- 泛型类中的泛型方法形参字符可以与泛型类相同，但是可以代表完全不同的类；静态方法不可以使用泛型类的泛型，需要将自身定义为泛型方法；基本指导原则为，如果能做到就尽量使用泛型方法而不是泛型化整个类
- `<? extends Number>` 上界，即传入类型必须为 Number 子类；`<? super Integer>` 下界
- 在java中是**"不能创建一个确切的泛型类型的数组"**的 `List<String>[] ls = new ArrayList<String>[10]` × `List<String>[] ls = new ArrayList[10]` √，`List<?>[] ls = new ArrayList<?>[10]` √ 如果引入泛型数组，那么因为泛型擦除机制① Coll和 ②Coll**会被看作同一种类型，那么我就可以①的数组转为 Obj 数组，让后把 ② 放进去，因为 ① 里的东西取出来都是 ① 类型的，那么那个 ② 取出来的时候也会变成 ①，由于泛型擦除，这是可以的，但是取出来之后调用方法时，内部的类型转换是 B 而不是 A, 将返回值赋值时就会出现问题。**最主要的是单独一个对象的时候你没法通过 Object 欺骗编译器使 Coll和 Coll看作同一种类型**
- **1.检查泛型的类型，获得目标类型 2.擦除类型变量，并替换为限定类型（T为无限定的类型变量，用Object替换） 3.调用相关函数，并将结果强制转换为目标类型。 `Pair<T extends Number>`，擦除后，类型变量用Number类型替换。**

CAS Compare and Swap

https://blog.csdn.net/v123411739/article/details/79561458

https://blog.csdn.net/qq_32998153/article/details/79529704

- **CAS 操作需要三个操作数：内存地址 V，旧的预期值 A,即将更新的目标值 B；当且仅当内存地址 V 的值与预期值 A 相等时，将内存地址 V 的值修改为 B，否则就什么也不做，整个操作是原子操作 （原子性操作是指在读改写是一个完整的操作，读过后不代表操作的完成）**
- **CAS 发现 V 中值与 A 不等时会尝试重新读取计算，这个重新尝试的过程称为自旋；其使用 volatile 保证读取的为最新的值**
- **CAS 循环开销很大，一直失败会带来很大开销；只能保证一个共享变量的原子操作**
- **为了解决 ABA 问题，可以使用版本号来解决（但实际上是根据需求来解决，如果 ABA 会导致 Error 时才需要解决）**
- **CAS 底层使用 unsafe 提供的原子性操作方法，Java 不能像 C/C++ 直接访问底层操作系统，而 unsafe 是 JVM 中的后门**

ViewRoot

参考《Android 开发艺术探索》

- **ViewRoot 对应 ViewRootImpl 是 WindowManager 和 DecorView 的纽带；在 ACThread 中，当 Act 创建完成后会将 DecorView 添加到 Window 中，同时会创建 ViewRootImpl 对象并和 DecorView 建立联系，View 的绘制流程都是从 performTraversals 方法开始的。**
- **在 measure 操作后，就可以通过 getMeasuredWidth/H 获取宽高，而且除特殊情况外就等同于 View 最终的宽高，layout 后可以获得 View 的四个顶点，通过 getWidth/H 获得最终宽高.**
- **DecorView 作为顶级 View 一般为一个竖向的线性布局，上部为 titlebar，下部为 content，所以 onCreate 中的方法为 setContentView，实际上我们的 View 被加到了 id 为 content 的 FrameLayout 中；我们可以 findId 取到 content ， content.childAt(0) 就是我们的 View.**

- 而我们的事件传递都会经过 DecorView 再传递到我们的 View

MeasureSpec

参考《Android 开发艺术探索》

- MeasureSpec 是一个 32 位的 int，高两位代表 SpecMode, 有 3 种模式：UNSPECIFIED，EXACTLY(父容器已精确计算出宽高), AT_MOST(父容器给出最大值); Decor 的 MSp 由窗口宽高和 LayoutPars 决定，普通 View 的 MSp 由 LPs 和父容器的 LPs 决定；MSp 代表的就是父容器对子 View 施加的约束规则
- 子View 的 MSp 和父容器的 MSp，自身的 LPs，margin，padding 有关
- 当 View 自身采用固定宽高的时候，不管父容器是什么，View 的 MSp 都是 EXACTLY，其精确遵循设定的大小；当 View 是 match_parent 时，View 的 MSp 是 EXACTLY, 且精确遵循父容器剩余的空间大小; 当 View 是 wrap_content 时，其总是 AT_MOST，且不会超过父容器的剩余空间，但 specSize 为 parentSize
- UNSPECIFIED 主要用于系统内部多次 Measure，不需要关注此模式

Measure

- measure 为 final 不可重写，onMeasure 中可以看出 View 的宽高就是 specSize，而 specSize 由 getSuggestedMinimumWidth 返回；如果没有背景返回 android:minWidth，如果有背景返回 minWidth 和 背景最小宽度中的最大值
- 如果我们继承了 View，那么我们要重写它的 onMeasure 方法，否则 wrap_content 和 match_parent 相同；因为在 wrap_content 模式下，它的宽高等于 specSize 而根据上一个可知为 parentSize, 而 parentSize 为当前父容器中可以使用的大小；我们可以重写 onMeasure，在 wrap_content 时 `setMeasuredDimension(w, h)` 设置我们自己的值，具体如何设置可以参考 TextView,ImageView 等做了特殊处理的 View
- ViewGroup 没有重写 onMeasure 方法，ViewGroup 是一个抽象类，它提供了 measureChild 方法，思想是 取出子元素的 LPs, 在通过 getChildMeasureSpec 创建子 View 的 MSp 传给子 View 的 measure 方法；VP 的具体 onMeasure 需要各个子类去具体实现
- 如线性布局，其中遍历了子 View，子 View 每 measure 后，VP 的 totalLength+, 子元素测量后，会测量自己的；布局如果是 match 则测量过程和 View 一样，如果是 wrap，则高度为 View 之和但是也不会超过父容器剩余空间。在 measure 之后就可以 getMeasuredH/W 获取宽高，但最好在 onLayout 中拿。
- View.post，或者 onWIndowFocusChanged 回调，但是会被回调多次，和 onResume，onPause 周期类似， 在 onStart 中 `view.getViewTreeObserver.addOnGlobalLayoutListener()` View 树状态或者内部 View 的可见性发生改变都会被回调；View.measure，match 则不可手动测量，因为我们不知道父容器剩余空间，具体数值可以构造 MSp 传入，wrap 时我们可以传入最大值来构造 View，（1 << 30）-1

Layout

- layout 方法中会先通过 setFrame 确定 View 的四个顶点位置，而后调用 onLayout，其中会遍历调用子 View 的 layout 方法
- onLayout 方法需要不同 View 自己去实现，对于线性布局 Vertical，其中会调用子 View 的 setChildFrame，其中仅仅就是调用子 View 的 layout，指定子 View 位置，同时 childTop 会逐渐增大，正好符合后加入的 View 会出现在下面
- setChildFrame 传入的宽高实际上就是 View 的测量宽高

Draw

- backgound.draw(canvas)（不可重写） -> onDraw -> dispatchDraw -> onDrawScollBars(还有前景)
- 存在 setWillNotDraw 标志位知指明该 View 不会绘制 UI，可以进一步进行优化，ViewGroup 默认为真，View 默认为假

自定义 View 须知

- 需要在 onMeasure 中对 wrap 做处理，否则无效果
- 需要在 draw 和 VP 的 onMeasure 和 onLayout 中处理 padding 和 margin
- View 中自带 post，不需要使用 handler
- 需要在 onDetachedFromWindow 中结束掉线程或者动画
- View 带有滑动嵌套的时候，需要处理好滑动冲突

GC

https://blog.csdn.net/weixin_36027342/article/details/79973294

https://blog.csdn.net/u010651249/article/details/83899049

- new，clone，反序列化，反射 创建对象
- GC 主要在堆中执行，堆中内存分为新生代和老年代，一般新生代内存大
- GC 主要关注哪些需要回收，什么时候回收，如何回收
- 引用计数法(无法解决循环引用问题，主流 JVM 不使用)；GCRoots 不可达或者说 强引用不可达
- 需要回收的对象会放在一个 F-Queue 中，在 Finalizer 线程中销毁
- 如何回收：标记清除（效率低会产生大量内存碎片）；复制算法（将可用内存分为两部分，每次只是用一块，内存使用率低）；标记整理算法（标记后移到一端，直接回收边界外内存）
- 现代商业虚拟机如 hotspot 使用复制算法，新生代 80% Eden，两个 Survivor 10%, 每次将 Eden 和 Survivor 中存活的拷贝到另一个 Survivor 中，清理 Eden 和 Survivor，如果不够就占用老年代内存
- JVM 中是分代收集算法，新生代中采用复制，老年代中采用标记整理
- （分为新生代和老年代，新生代内存不足先 Minor GC；移一部分到老年代；Major GC 释放老年代；OOM）
- 过大的对象会直接放入老年代

GC Root

- Class - 由系统类加载器（system class loader）加载的对象，这些类是不能够被回收的，他们可以以静态字段的方式保存持有其它对象。我们需要注意的一点就是，用户自己定义的类加载器并不是 roots
- Thread - 活着的线程
- 方法区中类静态引用，方法区中常量引用
- Stack Local - Java 方法的 local 变量
- JNI Local - JNI 方法的 local 变量
- JNI Global
- Monitor Used - 用于同步的监控对象
- Held by JVM -用于 JVM 特殊目的的 GC 保留对象

GC 收集器

- 新生代
  - Serial （单线程且工作时会暂停其它所有线程，但是由于新生代内存空间不大，速度很快效率很高，只可以与老年代的CMS和serial old GC一起工作 ）
  - ParNew （Serial 多线程版本，工作时也会暂停其它线程，只可以与老年代的CMS和serial old GC一起工作 ）
  - Parallel Scavenge（具有自适应调节策略，它会调节吞吐量和新生代内存大小，影响 GC 频率）
- 老年代
  - Serial Old （单线程，使用整理标记算法）
  - Parallel Old (整理标记算法，CPU资源敏感场景，可以考虑 Parallel Scavenge + Parallel Old)
  - CMS（Concurrent Mark Sweep，标记清除，尽量缩短停顿时间，只有初始标记和重新标记会 stop world，由于清理时，用户线程还在运行，会产生浮动垃圾导致失败，从而进行 full GC，且不能内存满时才进行回收，因为程序运行需要内存）
  - G1收集器 （很前沿，充分利用多 CPU，多核来缩短 stop the world 时间，使用G1收集器时，它将整个java堆划分为多个大小相等的独立区域，但是新生代和老年代不再是物理隔离了，它们都是一部分Region（不需要连续）的集合。G1优先回收价值最大的Region（有限时间内获取尽可能高的效率）
- jdk9及更新的版本中默认的是G1收集器；jdk8默认收集器：新生代GC：Parallel Scanvage 收集器；老年代使用：parallel old收集器

TCP & UDP

- TCP 面向链接，传输可靠，应用于传输大量数据，速度慢
- TCP 建立链接三次握手，客户端发起请求，服务器接收到再发送确认，客户端接收到再次发送，两端都进入 ESTABLISHED 状态
- HTTP 1.1 就是只建立一次 TCP 链接而重复的使用它传输一系列的请求，减少链接次数和开销
- TCP 和 UDP 是传输层，UDP 收发数据不建立链接，不会确认，所以不保证达到率，不保证到达顺序

App 测试

https://www.cnblogs.com/xiaohuhu/p/10267314.html

- 兼容性测试，Android端20多种，ios较少
- 压力/性能测试
- 手机操作系统，Android较多，ios较少且不能降级，只能单向升级；新的ios系统中的资源库不能完全兼容低版本中的ios系统中的应用，低版本ios系统中的应用调用了新的资源库，会直接导致闪退（Crash）；
- 操作习惯：Android，Back键是否被重写，测试点击Back键后的反馈是否正确；应用数据从内存移动到SD卡后能否正常运行等；
- push测试：Android：点击home键，程序后台运行时，此时接收到push，点击后唤醒应用，此时是否可以正确跳转；ios，点击home键关闭程序和屏幕锁屏的情况（红点的显示）；
- 安装卸载测试：Android的下载和安装的平台和工具和渠道比较多，ios主要有app store，iTunes和testflight下载；
- 升级测试：可以被升级的必要条件：新旧版本具有相同的签名；新旧版本具有相同的包名；有一个标示符区分新旧版本（如版本号），对于Android若有内置的应用需检查升级之后内置文件是否匹配（如内置的输入

法）
- 并发（中断）测试：闹铃弹出框提示，另一个应用的启动、视频音频的播放，来电、用户正在输入等，语音、录音等的播放时强制其他正在播放的要暂停；
- 数据来源的测试：输入，选择、复制、语音输入，安装不同输入法输入等；
- push（推送）测试：在开关机、待机状态下执行推送，消息先死及其推送跳转的正确性；应用在开发、未打开状态、应用启动且在后台运行的情况下是push显示和跳转否正确；推送消息阅读前后数字的变化是否正确；多条推送的合集的显示和跳转是否正确；
- 分享跳转：分享后的文案是否正确；分享后跳转是否正确，显示的消息来源是否正确；
- 触屏测试：同时触摸不同的位置或者同时进行不同操作，查看客户端的处理情况，是否会crash等

自动化测试

https://blog.csdn.net/eclipsexys/article/details/45622813/

https://blog.csdn.net/daihuimaozideren/article/details/78331673

- MonkeyRunner (Python); Instrumentation(基于单个 Activity)；QTP(web 上的通过录制脚本实现)；UiAutomator(基于 Andoid 4.x,无需签名，无任何 Activity 限制)
- UIAutomator 不支持 Hybird App，WebApp；2.0基于 Instrumentation, 可以获取 Context, 可以使用 Service 及接口；基于 Junit4，测试用例无需任何父类，方法名不限
- 通过声明 @RunWith（AndroidJUnit4.class），来表示该类为一个测试集合
- 通过声明 @Test，来表示该方法为一个测试用例
- `aapt dump badging C:\D\proj\HugoDemo\app\HugoDemo.apk >log.txt` 可以获取 LaunchActivity

测试用例

- 最顶层为 TestSuite 测试套件，下一层为 TestCase，测试用例集，其中包含多个测试用例
- 用例要有独立性，另外一定要有断言，因为用例是由期望结果的，需要知道测试结果
- 测试要注重测试覆盖率和测试效率，杀虫剂悖论，二八原则
- 测试按阶段分为单元测试，集成测试，系统测试，验收测试

ANR

https://blog.csdn.net/jaychou_maple/article/details/78782822

- Android 中 响应能力是由 Activity Manager 和 Window Manager 系统服务来监控的
- 只有主线程会出现 ANR，必须触发了某些操作或者调用了某些函数，而响应超时，在不同 Context 下，时间不同
- ANR 时系统会弹出对话框，使用 `adb pull /data/anr` 可以将 traces.txt 文件中的记录输出
- 在进行相关操作调用 hander.sendMessageAtTime() 发送一个ANR的消息；进行相关的操作；操作结束后 remove 掉该条 message。如果相关的操作在规定时间没有执行完成，该条message将被handler取出并执行，就发生了ANR。
- 主线程大量 IO 导致阻塞，包括网络和文件；其它进程占用 CPU 导致本进程得不到 CPU 时间片，如其它应用频繁读写；硬件资源操作，调用 Camera；线程 wait，或者争夺锁，其他线程崩溃导致主线程一直等待，或者死锁；service binder 数量达到上限；
- BroadcastReveiver 中的耗时操作交给 Service 来做，或者新开线程，用 Handler 处理与 UI 线程的交互

Crash

https://www.cnblogs.com/yoyoketang/p/9101365.html

- 运行内存不足，内存泄漏导致 OOM；内存越界，下标越界，逻辑错误等
- 权限不足；执行异常（非受检异常）抛出没有捕获
- 设备兼容性问题，kotlin 调用 java 参数非空
- 网络速度过慢导致 app 崩溃
- 解决 Crash 一是查看 log，二是去 sd 卡查看保存的日志 `adb logcat | find "com.sankuai.meituan" >d:\hello.txt` , 三是 bugly 手动上报

组件化

https://blog.csdn.net/nnmmbb/article/details/82969995

依赖管理

https://www.jianshu.com/p/f34c179bc9d0

- *.jar：只包含了 class 文件与清单文件，不包含资源文件，如图片等所有 res 中的文件。
- *.aar：包含所有资源，class 以及 res资源文件全部包含
- 如果你只是一个简单的类库那么使用生成的 .jar 文件即可；如果你的是一个 UI 库，包含一些自己写的控件布局文件以及字体等资源文件那么就只能使用 .aar 文件。
- 直接依赖第三方开源库，一般是托管在 jitpack 或者 jcenter；二是直接依赖本地 arr 文件，一般是在 libs 目录下；三直接依赖本地 jar 包；四依赖本地 module
- jar包：远程依赖不会打包到 arr，本地依赖会；arr：远程和本地都不会打包到 arr；如果要提供我的库工程的 arr 给他人，需要告诉他们远程 jar 和 arr（因为他们没有打包到 arr 中）
- provided（Compile only） 只在编译生效，不会打包到 apk 或者 arr 中；compile，api 会打包到 apk 或者 arr
- implementation 编译的依赖，它仅仅对当前的 Module 提供接口；使用 implementation 一可以提高编译速度，二可以对外隐藏不必要的接口
- 使用 `exclude group:'com.android.support` 排除 group, 使用 `exclude group:'',module:'design'` 指定具体

注解

https://blog.csdn.net/qq5132834/article/details/79156620

- 注解也叫元数据, 可以生成文档；编译检查；编译时动态处理生成代码；运行时动态处理例如用反射注入实例
- 注解分为标准注解，元注解，自定义注解；元注解包括 @Retention 用于标明注解被保留的阶段，@Target 用于标明注解使用的范围，@Inherited 用于标明注解可继承，@Documented 用于标明是否生成 javadoc 文档。
- class 文件是严格有序的，对于类字段，方法等编译器会将对应的注解信息存放到他们对应的 attributes 中
- 当 Main 类被编译时，对应的 Class 中会存在一个 RuntimeVisibleAnnotations 属性，当该字节码文件被加载时对应的 RuntimeVisibleAnnotations 属性值会存到 Main.class 中

- 编译后注解就是一个继承了 Annotation 的接口类型，当我们调用 `Main.class.getAnnotation(Test.class)` 时，会通过动态代理生成一个 Test 接口的实现对象，并把注解的 value 内容存入，我们就可以通过 value 方法获取到注解的值
- 通过反射处理注解会有性能问题所以现在很多框架都是编译时处理如 Dagger2

Dagger2 入门

https://www.cnblogs.com/all88/p/5788556.html

https://www.jianshu.com/p/626b2087e2b1

- Dagger 是一个依赖注入框架，比如 Activity 中会用到很多类，那么 Activity 就会依赖很多类，我们可以创建一个容器 Ioc，把类代理给它，Activity 只需要依赖这个容器；Dagger 就可以创建这个容器
- @Inject 注解一个成员表示 Dagger 会为它提供依赖
- @Inject 注解一个构造函数表示它会为 Inject 修饰的成员变量提供依赖；@Provided 注解一个方法同上，多用于第三方库构造方法无法修改；@Module 注解一个类表示可以来该类内寻找依赖；@Component 指定去哪里寻找依赖
- @Component 一般用来标注接口，被标注的接口编译时会产生相应的类实例来作为供需双方间的桥梁
- 流程
    - 1.查找 Module 中是否存在创建该对象的方法
    - 2.不存在参数直接创建，否则递归 1
    - 3.Module 中不存在则遍历 Inject 构造函数
    - 4.同 2
- *.java -> Parse and Enter -> APT(生成一些 *.java 文件) -> Analysis an Generate -> *.class

---

Retrofit

https://www.jianshu.com/p/07f7eb4aa9ae

- 用动态代理的方式，在调用接口方法前后注入自己的方法，before 通过接口方法和注解生成网络请求的 request，after 通过 client 调用相应的网络框架发起请求，并将 response 通过 converterFactory 转换为相应的 model
- loadServiceMethod 会对 method (即我们定义的接口方法) 进行处理，构成 ServiceMethod 对象；Retrofit 内部使用 serviceMethodCache 这个表来缓存 serviceMethod 如果表内没有才会通过 Builder 创建；Builder 中的大部分参数都是接口方法中定义的或者是通过注解反射获得的，没有真正传入参数只是对参数类型做了解释，参数是传给 OkHttpCall 的
- ServiceMethod 与接口方法传入的参数以统传入 OkHttpCall，构成一个 Call 对象；`okHttpCall = new OkHttpCall<>(serviceMethod,args)` ，`return servceMethod.callAdapter.adapt(okHttpCall)`
- 本质上 OkHttpCall 是对 OkHttp 的封装，调用的是 enqueue 方法，在代理对象 invoke 返回时有 adaptCall 步骤

动态代理

https://www.cnblogs.com/gonjan-blog/p/6685611.html

- 动态代理通过 `Proxy.newProxyInstance` 传入一个 InvocationHandler 来创建；InvocationHandler 中要持有真实对对象的引用
- 动态代理的好处是，可以统一管理所有代理方法，因为所有方法都会调用 invoke 函数, newProxyInstance 只是封装了创建代理类的流程，真正产生对象的方法是内部的 `Class<?> cl = getProxyClass0(loader, intfs)`
- 可以反编译生成的代理类 class 文件，其继承了 Proxy，实现了 Person 接口; 该代理类的构造方法需要传入 Handler 对象，而 InvocationHandler 对象又持有真实对象引用；该类中有一段静态代码块其中通过 `Class.forName("").getMethod()` 反射获得了接口中的方法，equals，hashcode，toString 等方法；在代理类的对应方法中会有 `this.h.invoke(this,m3,null)` 调用了 Handler 中的方法，并且把 method 对象传入；在 Handler 的 invoke 方法中会执行 `method.invoke(target,args)`

Handler

https://blog.csdn.net/wsq_tomato/article/details/80301851

https://blog.csdn.net/wsq_tomato/article/details/80893990

- Message：线程间传递的消息，内部可以携带少量信息字段，可以携带 obj; Handler：基本上都是在调用 `sendMessageAtTime` 除了 `sendMessageAtFrontOfQueue`；Message: 每个线程只有一个 MessageQueue; Looper: 每个线程只能绑定一个 Looper，否则 prepare 时会报错，loop() 后开始死循环，从 Queue 中取 Msg 传递给 HandleMsg；ThreadLocal：用来保存 Looper 和 Queue，可以保证每个线程只有一个 L 和 Q, 是一个线程内部的数据存储类
- 在 prepare 里对 ThreadLocal 判空并 `sThreadLocal.set(new Looper(quitAllowed))`；在 Looper 构造造函数里 `mQueue = new MessageQueue(quitAllowed)`；主线程在 main 方法中调用了prepare 和 loop
- loop 中创建了死循环，取除 msg 不为空则 `msg.target.dispatchMessage(msg)`
- 在 dispatch 中如果我们设置了 Callback 就会直接调用 handleCallback，如 runOnUiThread 中 `mHandler.post(action)` 和 `action.run()`
- 如果 Handler 是非静态内部类，则会持有外部类引用，当其它线程持有了该 Handler 可能会造成内存泄漏；MessageQueue 中存在未处理完的 Msg，Msg 的 target 也持有 Ac 的引用
- 使用静态内部类 + 弱引用减少内存泄漏；在外部类生命周期结束时清空 Queue
- 在 obtain 方法中，Msg 是一个链式存储结构，返回 sPool 指向的对象，并把 sPool 后移；处理后的消息会被recycle 重置属性后加到 sPool 后，头插法
- 在加入 Queue 时会根据时间点大小判断需要插入的位置，同时还需要判断时候需要去唤醒主线程发送当前队首的消息；Queue 中的 msg 是按找发送时间点从小到大排列的

Binder

- 从 IPC 角度来说，Binder 是进程通讯方式，可以理解为一种虚拟的物理设备；从 Android Framework 角度来说，它是 ServiceManager 链接各种 Manager 和相应 ManagerService 的桥梁；从应用层来说 BInder 是客户端和服务端进行通讯的媒介。
- 普通的 Service 中的 Binder 不涉及进程间通讯，所以我们说 AIDL（Messenger 底层也是 AIDL），在 AIDL 自动生成的 java 文件中会有一个内部类，Stub，在一个进程间通讯的时候没什么用，跨进程的时候就会通过 Stub 内部的 Proxy 代理来实现，它会走 tansact 方法
- asInterface 方法，将服务端的 Binder 对象转换为客户端所需的 AIDL 接口类型的对象，该方法会区分进程，同一进程返回 stub 对象，否则返回 Stub.proxy

- onTransact 方法运行在 Binder 的线程池中，通过 code 确定目标方法，data 取出参数，返回值写入 reply；返回 false 代表请求失败，可以做权限验证
- Proxy.getBook 方法运行在客户端，调用时传入 Parcel 的 data 和 reply，接着调用 transact 方法发起 RPC (远程过程调用) 请求；同时当前线程挂起；然后服务端的 onTransact 会被调用，知道 RPC 返回当前线程继续执行取出 reply
- 远程请求耗时的话不能在主线程发起请求，同时 Binder 方法应该是同步的，因为 onTransact 方法已经在线程池中了
- binder.linkToDeath 传入一个 DeathRecipient 可以监听 Binder 的死亡，重新唤起链接

Android 65535

https://www.cnblogs.com/android-blogs/p/5778997.html

- 因为在 Dalvik指令集里，调用方法的 invoke-kind 指令中，method reference index 只有 16位，class，field 一样
- 上层十几个业务线为独立module，都依赖base，而 base 的资源id有个三四千，上层R文件会把下层的R文件合并过来，使用 multidex 后，会把 manifest 里的 activity、service 等和其直接引用类加到 main dex 中，所以很多R文件涌入，field 超个65535
- 检查您的应用的直接和传递依赖项 ，通过 ProGuard 启动代码压缩移除未使用的代码
- 对 app 进行分包

事件分发机制

https://www.sgxm.tech/?p=336

https://www.jianshu.com/p/d3758eef1f72

双亲委派

https://blog.csdn.net/wangyang1354/article/details/49448007

- 委托机制可以繁殖内存中出现重复的字节码
- 当前ClassLoader首先从自己已经加载的类中查询是否此类已经加载，如果已经加载则直接返回原来已经加载的类。
- 每个类加载器都有自己的加载缓存，当一个类被加载了以后就会放入缓存，等下次加载的时候就可以直接返回了。
- 当前classLoader的缓存中没有找到被加载的类的时候，委托父类加载器去加载，父类加载器采用同样的策略，首先查看自己的缓存，然后委托父类的父类去加载，一直到bootstrp ClassLoader.
- 当所有的父类加载器都没有加载的时候，再由当前的类加载器加载，并将其放入它自己的缓存中，以便下次有加载请求的时候直接返回
- 我们JDK本生提供的类库，比如 hashmap, linkedlist 等等，这些类由 bootstrp 类加载器加载了以后，无论你程序中有多少个类加载器，那么这些类其实都是可以共享的，这样就避免了不同的类加载器加载了同样名字的不同类以后造成混乱。

基于 LRUCache 的缓存

- 对于图片来说我们分为强引用和软引用缓存两个部分，都采用 LRU，将引用放到 LinkedHashMap 中，初始化时会设置缓存空间大小
- LRU（Least Recently Used ）缓存满的时候删除最近最少使用的，其会每次调用时将访问的数据放到队列尾部，则头部为没有访问的数据，LRUCache 使用 LinkedHashMap，且 accessOrder = true. LruCache 缓存在内存中
- Bitmap 优化的核心就是采用 BitmapFactory.Options 来加载，inSampleSize 参数决定了采样率，一般为 2 的指数 2，4，8 ... 加载前预估图片所需内存，展示控件的实际大小，屏幕的分辨率

待看，反射，注解, Retrofit, ConcurrentSkipListMap & 自旋锁

单例

```
//DCL
class KLazilyDCLSingleton private constructor() : Serializable {//private constructor()构造器私有化

    fun doSomething() {
        println("do some thing")
    }

    private fun readResolve(): Any {//防止单例对象在反序列化时重新生成对象
        return instance
    }

    companion object {
        //通过@JvmStatic注解，使得在Java中调用instance直接是像调用静态函数一样，
        //类似KLazilyDCLSingleton.getInstance(),如果不加注解，在Java中必须这样调用：KLazilyDCLSingleton.Companion.getInstance().
        @JvmStatic
        //使用lazy属性代理，并指定LazyThreadSafetyMode为SYNCHRONIZED模式保证线程安全
        val instance: KLazilyDCLSingleton by lazy(LazyThreadSafetyMode.SYNCHRONIZED) { KLazilyDCLSingleton() }
    }
}
```

```
//静态内部类
class KOptimizeSingleton private constructor(): Serializable {//private constructor()构造器私有化
    companion object {
        @JvmStatic
        fun getInstance(): KOptimizeSingleton {//全局访问点
            return SingletonHolder.mInstance
        }
    }

    fun doSomething() {
```

```
        println("do some thing")
    }

    private object SingletonHolder {//静态内部类
        val mInstance: KOptimizeSingleton = KOptimizeSingleton()
    }

    private fun readResolve(): Any {//防止单例对象在反序列化时重新生成对象
        return SingletonHolder.mInstance
    }
}
```

排序算法

```
void quick(int *arr,int left,int right){
    if(right < left) return;
    int p = arr[left];
    int i = left, j = right;
    while(i < j){
        while(arr[j] >= p && i < j){
            j--;
        }
        a[i] = a[j];
        while(arr[i] <= p && i < j){
            i++;
        }
        a[j] = a[i];
    }
    arr[j] = p;
    quick(arr,left,j-1);
    quick(arr,j+1,right);
}
```